Spring 2019

Self-configured Elastic Database with Deep Q-Learning Approach

Yifu Zhou
yifuzhou@iastate.edu

Self-configured Elastic Database with Deep Q-Learning Approach

by

Yifu Zhou

A Creative Component submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to my major professor Dr. Zambreno and to my roommate Jie Yuan without whose support I would not have been able to complete this work. I would also like to thank my friends and family for their loving guidance and financial assistance during the writing of this work.

ABSTRACT

Elastic databases have grown in popularity over conventional databases in recent years due to their ability to be allocated with sufficient capacity for peak load. Especially with the support of the cloud platform, which provides flexible resources and low cost, elastic databases on the cloud show their excellent potential in scalability, flexibility, and accessibility. However, the interaction between the cloud layers of virtual machines (VMs) and databases further complicates the issue of cloud configuration to adapt to dynamic workloads. In this paper, I explore a framework for a self-configured elastic database that can optimize the cloud configuration and adaptively allocate resources under the constraints of databases' Service Level Agreement (SLA). At the core of the framework is a Deep Q learning approach, which combines the advantages of Reinforcement Learning (RL) and Deep Learning (DL). The framework is built on Amazon Web Service (AWS)'s cloud environment and uses MySQL database for its high availability replication mechanism. Experimental results on the TPC-W benchmark demonstrate that with the implementation of Deep Q learning, the elastic database reduces SLA violation by more than 90%, in the response to the steep slope of workload change.

CHAPTER 1. INTRODUCTION

1.1 Background

In modern usage Cloud Computing, known as the large multi-tenant hosted cloud platforms, are required to scale to millions of applications with unpredictable workload. In such complex environments, elastic databases on the cloud play an important role in providing *elasticity*, the ability to scale out during high load and scale in during low load. However, because of the cloud's on-demand availability, there are two key challenges for elastic databases. First, system performance need to be assured under the constraints of a Service Level Agreement (SLA), where Service Level Objectives (SLOs) such as throughput and response time highly depends on user workload. Especially with such unforeseeable workload, elastic databases will misuse to execute scale out or scale in rules frequently. This behavior will reduce resource utilization and even put infrastructures in high risk. Second, elastic databases are built as distributed systems; data capacity overload or other unexpected reasons will cause servers to crash and the data in those servers will be dropped. Thus, how to coordinate the cloud layers of virtual machines (VMs) and databases is the key for the design of elastic databases to provide high performance and high availability.

1.2 Related Work

In the development of building distributed systems, it is hard to guarantee transaction with scalability and availability. Designers have to trade off the ability of distributed system and data transaction [10]. In traditional database management systems (DBMSes), ACID transactions [26] can access several databases' partitions. However, in a distributed system, transactions can become a major performance bottleneck [6]. One of the general ways to solve this problem is a design to move away from sustaining ACID transactions. In today's popular distributed systems, such as Facebook's Cassandra [11] and Amazon's Dynamo [8], ACID transaction support is not provided.

Other systems only provided limited ACID transactions. For example, Accordion [18] only supports ACID transactions in partition-base DBMSes. Microsoft's Azure [3] is limited to small subsets of data. Moreover, even though the design of some optimal databases can scale without sacrificing ACID guarantees, it is difficult to derive such designs for enterprise-class systems [15]. Elastras designs a two-level hierarchy of Transaction Managers to provide elastic scalability [7]. Swat uses swap-based load balancing algorithm to reduce the SLA violation [14]. Furthermore, the implement of machine learning is the tend to predict dynamic workload in distributed system [16]. Especially in scalability system, accurate forecast will effectively improve the performance of interaction with clients' operation. Wang et al. [24] proposes a two-layer control architecture that the secondary control loop could reduce server consumption effectively. Bu et al. [2] uses a model-free learning approach to coordinate virtual machine's parameter with clients' workload.

1.3 Our Work

In this paper, we explore a framework of self-configured elastic database with the implementation of Deep Q learning, which combines the advantages of reinforcement learning (RL) to make a better decision through the previous experience, and deep learning (DL) to solve the high dimensional states issue. Experimental results show that with the implementation of Deep Q learning, self-configured elastic database can reduce SLA violation by more than 90% in the response to the steep slope of workload change. Specifically, Figure 1.1 shows the construction of our self-configured elastic database is three-fold:

- Client Emulator Layer. This layer simulates the user interface and it is at the top-most level of a self-configured elastic database system. The main function of Client Emulator Layer is to customize a number of clients in order to test the response of the system. It also can set the ratio of read query and write query to fully simulate the real-world operating environment.

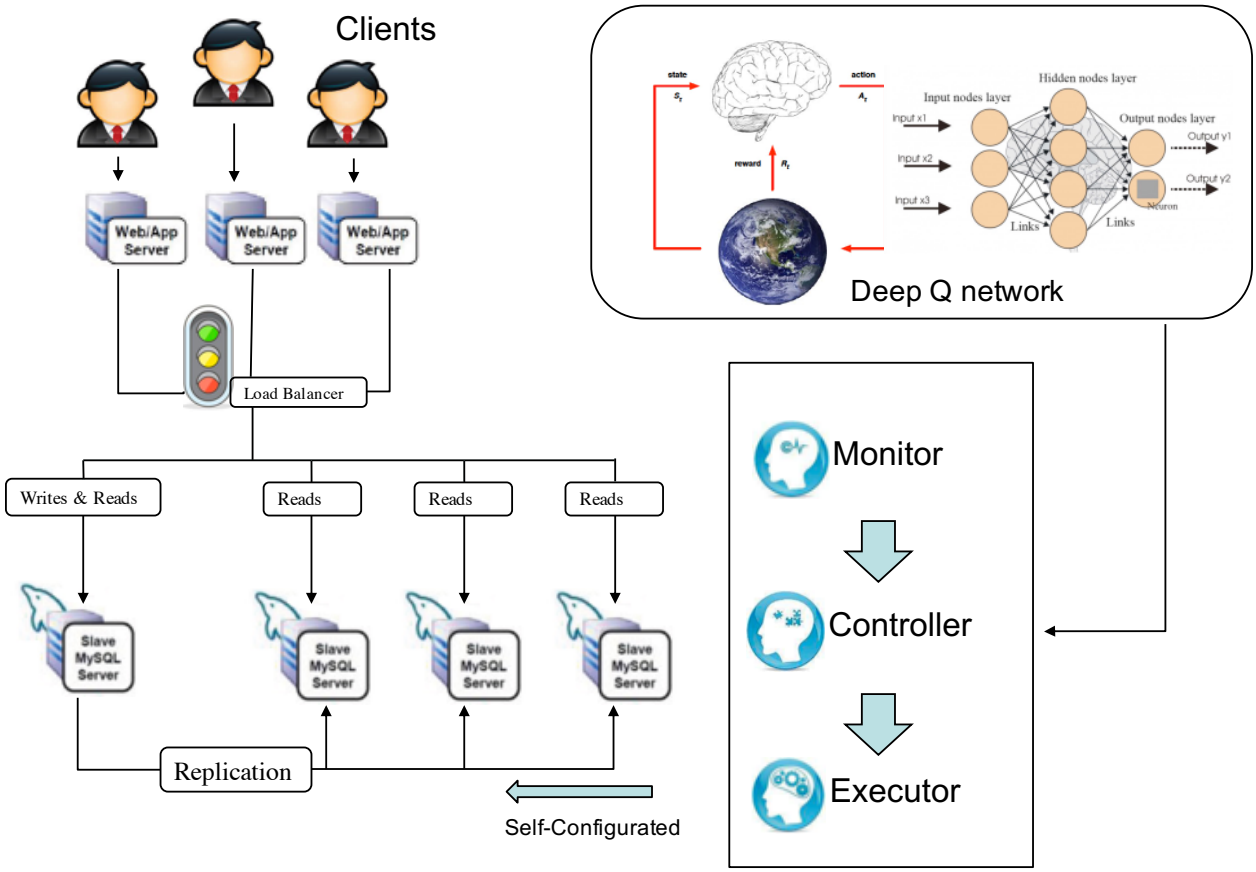

Figure 1.1 Self-configured Elastic Database Construction.

- System Logic Layer. This layer is the kernel of a self-configured elastic database, which is composed of four sections: load balancer, monitor, controller and executor. The effective interplay of these sections can keep the whole system at high performance.

- Database Layer. This layer is the server cluster mainly installed MySQL database on the Amazon Web Service (AWS). It is a master-slave architecture that enables data from master MySQL database to be replicated naturally to slave MySQL database keeping the whole system high availability.

The rest of the report is organized as follows: Chapter 2 provides a high-level description of a self-configured elastic database construction which features a three-layer design. Chapter 3 emphasizes the Deep Q Learning algorithm and how it impacts to a self-configured elastic database in our project. Chapter 4 provides a detailed evaluation and Chapter 5 concludes the report and demonstrates future challenges.

CHAPTER 2. SYSTEM MODELING

2.1 Client Emulator Layer

The Client Emulator Layer is the entrance of the elastic database. At the beginning, it allows the Client emulator to customize a sequence of workloads to access the database at a given time. For example, $U =< U_1, U_2, \cdots, U_n >$ represents the Client Emulator allocating U_1 user sessions at time 1, U_2 user sessions at time 2, and U_n user sessions at time n. For each user session, it has a workload vector V, representing the number of queries a user session takes from the query pool to access the database. At the same time, the Client Emulator will start the thread of the Controller and the Executor which belong to the System Logic Layer. In the user session, the Client Emulator will start U_{max} threads where U_{max} is the maximum number in workload U, so each thread represents a user session. At the beginning, all user sessions will be suspended. When a user session is activated, it will select to execute a read or write query based on the RW ratio, if the RW ratio is 1, it means that the query must be a read query, if the RW ratio is 0.5, it means that there is a 50% chance of read query and 50% chance of write query. With such a design, it is easy to test the system performance in different operating modes.

2.2 System Logic Layer

Figure 2.1 shows the interplay in the System Logic Layer. Based on the user session demands, the Monitor is responsible for collecting statistics from the system resource pool [5] such as virtual memory, CPU and network connections to periodically inform the Controller. The Controller is responsible for analyzing the performance of the elastic database and do a decision making, then let the Executor to adjust system configuration.

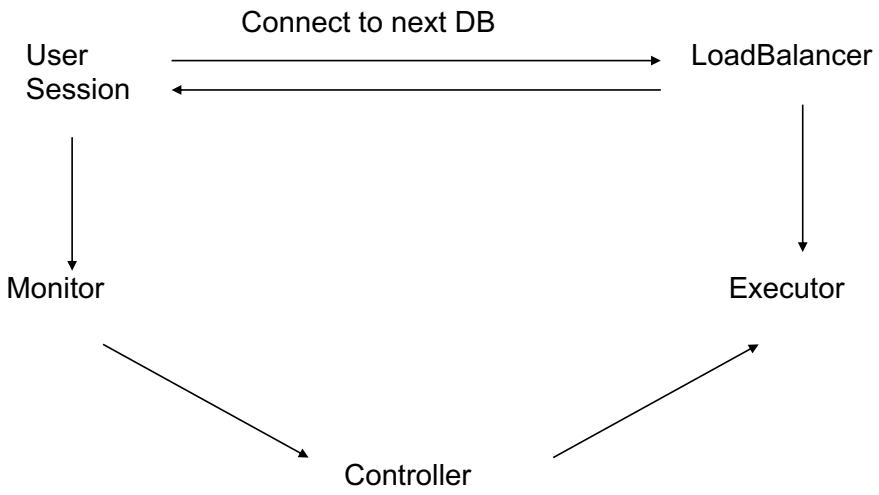

Figure 2.1 System Logic Layer.

2.2.1 Load Balancer

The Load Balancer is one of the key parts of the elastic database. The purpose of the Load Balancer is to optimize resource utilization. Especially, on the demand of high throughput and low response time in elastic databases [4], the Load Balancer can maximize throughput and minimize response time to avoid overload of any single resource. Besides, distributed system with a Load Balancer provides more availability in solution deployment than a standalone server. For example, it is easy for other backup databases be taken out of the load balancing pool if one database is down. The algorithm in the Load Balancer can be divided into two parts: stateless algorithm and state-based algorithm. Round-robin [19] is one of the most popular stateless algorithms that load balancer selects the next database connection in equal portions and in circular order. Least latency state-based algorithm is to consider the system resource and select the least latency database as the next connection. In the design of a self-configured elastic database, the cooperation between least latency state-based algorithm and Deep Q Learning lets the elastic database operate efficiently.

2.2.2 Controller

The Controller is another important component of a self-configured elastic database. It is the carrier of the Deep Q Learning algorithm which is the heart of a self-configured elastic database. According to Deep Q Learning, at the state of current environment of the system, the Controller chooses an action which is to configure the cloud environment. After that, the Controller will receive feedback information as a reward, and then transforms to a new environment state. Therefore, Deep Q Learning can figure out an optimal configuration policy in finite steps. The detail of Deep Q Learning will be described in chapter 3.

2.3 Database Layer

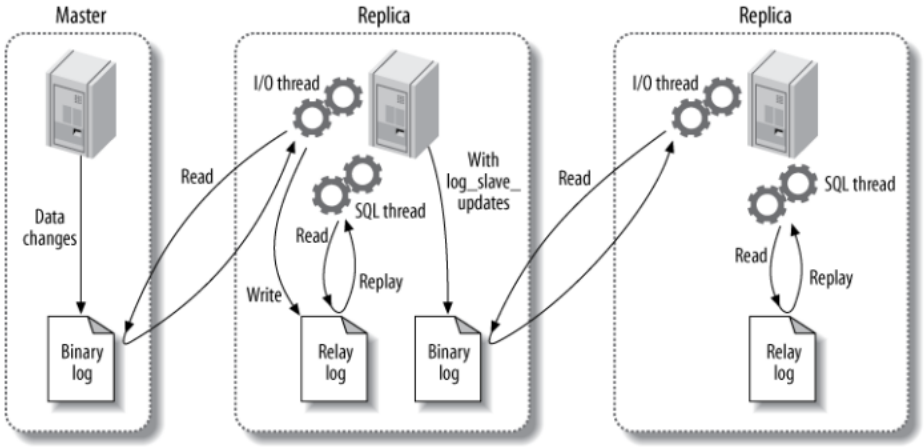

Figure 2.2 MySQL Master-Slave structure. (From [28])

This layer is the bottom layer in a self-configured elastic database. The framework is built on the AWS platform and all VMs will be installed MySQL database making up a database cluster. It is a master-slave structure as shown in Figure 2.2. It has the property of Read/Write Splitting and high availability. The master server is in charge of write and read operation and the slave server is only responsible for read operation. When the client updates data on the master database, the master database will immediately follow executive command and the command is written to the local binary log. Furthermore, the master database will record the log ID. If slave databases are online,

the command will be transmitted to the slave's relay log in parallel through I/O thread. Otherwise, the slave databases will receive a list of all pending statements from the master database's binary log when the slave is online. Once the data is received in the slave's relay log, the slave SQL thread will execute the command locally, bringing the slave up-to-date with the master.

In conclusion, the Client Emulator Layer provides more operating modes to test the performance of our self-configured elastic database. The System Logic Layer provides an efficient power to drive the system to a high-productive level, and the Database Layer provides fault tolerance protecting the system.

CHAPTER 3. METHODS AND PROCEDURES

3.1 Self-configured Decision Making

In today's popular cloud computing environment [30], the coordination between cloud's infrastructure and its resident applications is very important [1]. Traditionally, the cloud system's parameters are configured manually. [27] Despite traditional design of cloud computing's inefficiency and inconvenience, resource balance is a critical problem [23]. Therefore, it is important to find an intelligent management method that can dynamically allocate cloud's resources. One of the most famous implementation is Reinforcement Learning (RL). RL is a trial-and-error learning process. An agent executes an action at current state transforming to next state and receives an immediate reward. After a sequence of interactions with the managed environment, an agent gradually learns a good policy, maximizing the overall expected reward. A RL problem is often generated as a finite Markov Decision Process (MDP) problem. It can be formulated with a set of states S, a set of actions A, the reward function $R_a(s, s^{'}) = E(r_{t+1}|s = s_t, s^{'} = s_{t+1})$, and the transition probability function $P_a(s, s^{'})$, which is the probability of choosing action a from state s to state $s^{'}$. Therefore, the process of MDP is:

s_0, a_0, r_0, p_0, s_1, a_1, r_1, p_1, s_2,$\cdots$,s_t, a_t, r_t, p_t, s_{t+1},$\cdots$,s_{n-1}, a_{n-1}, r_{n-1}, p_{n-1}, s_n

where s_t is the current state, a_t is the current action, r_t is the current reward of action a_t at state s_t.

Furthermore, the action selection at the current state not only depends on the current reward, but also depends on the further rewards at state s_{t+k} where $k = 1, 2, 3, \cdots$. Thus, the value function is proposed to estimate the future accumulated rewards:

$$Q(s, a) = E(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a) \tag{3.1}$$

where $0 < \gamma < 1$ is a discount factor and the optimal value function $Q^*(s, a)$ can be defined as the following:

$$Q^*(s,a) = E(\sum_{s' \in S} P_a(s,s')(R_a(s,s') + \gamma \max_{a'} Q^*(s',a')) \tag{3.2}$$

where s' and a' are the next state and the next action. For the reason that decision making is based on long-term rewards experience, so the agent can deal with local optimum problems. The purpose of RL is trying to find an optimal policy π^* that can achieve maximal total rewards from the initial state to the end state. This policy can be defined below:

$$\pi^*(s) = \arg\max_a (R_a(s,s') + \gamma \sum_{s' \in S} P_a(s,s') Q^*(s',a')) \tag{3.3}$$

According to this equation, the optimal policy is to select the best action which should be the maximum sum of an immediate reward and a discounted future reward of the next state. In the process of each iteration, RL will select an action on the current policy and receive a reward. The new reward will be used to update the value function. After sufficient iterations, the value function will converge to its optimal value and the optimal policy is formed.

In this work, we consider self-configured property in our elastic database as a RL problem. Environment is the interface with clients and agents are defined as the cluster of virtual machines in cloud. The state refers to the virtualized resource of VMs [29]. For our purposes, we consider CPU cores, threads per core, and a scale from 1 to 3 for memory usage as states. An action is defined as to one-unit alteration of a single resource. If the alteration exceeds the default CPU or memory requirement, elastic databases would run a scale out rule to add more databases. In order to simplify the RL problem, the state transition probability $P_a(s, s')$ is set as 1.

A reward value is defined to reflect the performance of the system. As we know, response time and throughput are two most important metrics in elastic databases. Referring to Bu et al. [2]'s reward function, let P and T represent throughput and response time respectively, P_{SLA} and T_{SLA} represent the corresponding service level agreements, and C_p and C_t represent

the penalty of SLA violation. Thus, the formula of reward function is defined as follows:

$$r = \frac{P}{P_{SLA}} + \frac{T_{SLA}}{T} - C_p - C_t, \tag{3.4}$$

where

$$C_p = \begin{cases} w_p * \frac{P_{SLA}}{P}, & P < P_{SLA} \\ 0, & otherwise, \end{cases} \qquad C_t = \begin{cases} w_t * \frac{T}{T_{SLA}}, & T > T_{SLA} \\ 0, & otherwise, \end{cases} \tag{3.5}$$

where w_p and w_t are weights of measurement the impact of SLA violations.

Q-Learning is one of the most famous value-based algorithms in the RL algorithm. $Q(s, a)$ is the value function in the state $s \in S$ and the action $a \in A$ at a certain time. The main idea of Q-Learning algorithm is using the state and the action to construct a Q-table which stores the Q value, and then the agent selects the action that can obtain the maximum benefit according to the Q value. Thus, we can learn with the Q value through the process of Q-table update. The update of Q-table can be defined below:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)], \tag{3.6}$$

where α is the learning rate and γ is the reward decay coefficient. The above formula is updated by Q-learning, and the largest $Q(s', a')$ will be selected according to the next state s'. The value of $Q(s', a')$ multiplied by the reward decay coefficient γ and plus the true return value is the realistic Q value, and $Q(s, a)$ in the previous Q table is used as the estimative Q value.

However, Sutton et al. [21] showed that Q-learning is only suitable when the state and the action space are discrete or low-dimension. When the state and the action space are high-dimension and continuous, such as in elastic databases, for each cluster, if there are n types of resources, and assume that each resource scale in m levels, so there is a total of m^n configuration states and the Q table should be used to store $n * m^{n+1}$ Q values without considering the property of *elasticity*. Therefore, it is too difficult using Q-table to store Q value for each state-action pair.

Bu et al. [2] used hybrid reinforcement learning approach with Simplex method to reduce high-dimensional parameter. In our self-configured elastic database, we convert Q-table update process into a function fitting problem [13]. The Q value can be generated by a function approximation which is a multi-layer neural network instead of a Q-table. The combination of Deep Learning (DL) with Reinforcement Learning (RL) is what is referred to as the Deep Q Learning (DQL), also known as Deep Q Network (DQN). DQL uses a reward value through Q-learning to construct labels. It also uses an experience replay mechanism to stores previous experiences. Since Q-learning is an off-policy offline learning method [25], it can learn the current experience, the previous experience, and even future experience. Consequently, randomly joining previous experience during the learning process will make the neural network more efficient. Moreover, for the reason that in RL, input data is time-sequential, which does not satisfy the requirement of independent identical distribution by neural networks. Experience replay [17] can solve the problem of correlation and non-static distribution. It stores the transfer samples likes (s_t, a_t, r_t, s_{t+1}) in each timestamp to playback memory network. It then randomly takes out some samples to disrupt the correlation. In DQL, we will use two neural networks with identical structure but different parameters. $EvalNet$ is the neural network using latest parameters to estimate Q value, while $TargetNet$ is the neural network using long ago parameters. $Q(s_t, a_t; \theta)$ represents the output of $EvalNet$ which is used to evaluate the value function of the current state-action pair; $Q(s_{t+1}, a_{t+1}; \theta^-)$ represents the output of $TargetNet$ which can get $TargetQ$:

$$TargetQ = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta^-), \tag{3.7}$$

According to the loss function:

$$loss = [TargetQ - Q(s_t, a_t; \theta)]^2, \tag{3.8}$$

the parameter of $EvalNet$ can be updated and at the end of every iteration, the parameter of $EvalNet$ will be copied to TargetNet [12]. Algorithm 1 shows our Deep Q Learning approach in detail.

Algorithm 1 Deep Q Learning in Self-configured Elastic Database.

Initialize replay memory D to capacity N;
Initialize action-value function Q with random weights;
repeat
    $s_t = get\_current\_state()$
    $a_t = get\_action(s_t)$ using self-configured exploration policy
    reconfigure(a_t)
    $r = observe\_reward()$
    $s_{t+1} = get\_current\_state()$
    $store\_transition(s_t, a_t, r_t, s_{t+1})$ to D
    Sample random mini-batch of $transitions(s_j, a_j, r_j, s_{j+1})$ from D
    if s_{j+1} terminal **then**
        $TargetQ = r_j$
    else
        $TargetQ = r_y + \gamma \max_{a_{y+1}} Q(s_{y+1}, a_{y+1}; \theta)$
    end if
    $loss = [TargetQ - Q(s_t, a_t; \theta)]^2$
until value function converges

3.2 Self-configured Exploration

Traditional Reinforcement learning uses a $\epsilon - greedy$ policy for action selection. This algorithm is the core in RL which starts from an unknown environment. $\epsilon - greedy$ is a trade-off decision of exploitation and exploration [22]. Exploitation is the selection based on current optimal policy; on the contrary, exploration is to select random actions. The purpose of random selection is to explore potential new environment to refine the current policy. Thus, the productivity of $\epsilon - greedy$ depends on the random selection with a probability ϵ, otherwise, $1 - \epsilon$ is the probability of exploitation. In RL algorithm, if the ϵ is so small that it will limit the agent to explore new environment and decelerate the learning process; in contrast, if the ϵ is so large that the learning process is too hard to learn from the optimal policy and make an inaccurate decision.

In our self-configured decision making, we have to refine existing $\epsilon - greedy$ algorithm to adapt to our environment. Refer to Bu et al.'s [2] knowledge guided exploration, We consider CPU utilization as the most performance relevant system metric and set U_{cpu}^{ub} and

U_{cpu}^{lb} as the upper bound and lower bound for CPU utilization, respectively. Once the CPU utilization is beyond this range, our self-configured exploration policy will amend an optimal action to satisfy the requirement of CPU utilization.

CHAPTER 4. EXPERIMENT AND RESULTS

4.1 Environment Setup

In this chapter, we designed experiments to evaluate the performance of a self-configured elastic database. The system is built on the Amazon Web Service (AWS) and utilized five instances of EC2 virtual machines. The default configuration of each EC2 VM has 4 vCPUs, 2.3GHz of Intel Broadwell E5-2686v4 processor, 16 GiB memory and 2 threads per core. All VMs run on the Amazon Machine Image (AMI) of Ubuntu Server 14.04 LTS (HVM). In our experiments, EC2 VMs are only used for providing servers' environment and would not use any optimized service from AWS. The benchmark I selected was TPC-W [9], which simulated an online bookstore scene to test the performance of hardware and software operating environments.

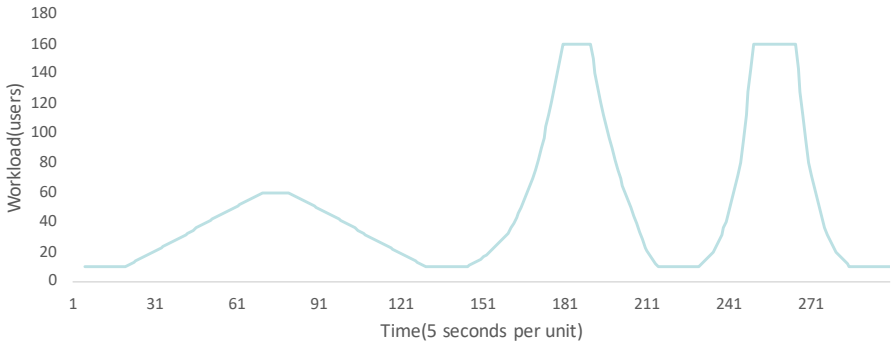

Figure 4.1 User Workload for Experiment.

The purpose of the experiment is to test how a self-configured elastic database performs when we simulate a sudden slope of the workload. Thus, in our experiments, we need to simulate a fluctuating workload to actuate decision making. According to the data statistics from Amazon website, which is the largest e-commerce marketplace in the world, the

maximum user increase rate is 50% per day in transaction. And the peak increase rate on holiday is up to 70% from 2016 to 2017 [20]. In our experiment, we fluctuate three kinds of workload environments refering to these data:

$$U_{t+1} = \begin{cases} U_t \pm 50\% * U_0, & time\ slot\ 1 \\ U_t \pm 50\% * U_t, & time\ slot\ 2 \\ U_t \pm 70\% * U_t, & time\ slot\ 3 \end{cases} \tag{4.1}$$

where U_0 is the initial workload input to the client simulator layer.

In our experimental workload, shown as Figure 4.1, the initial workload is set as 10, the maximum critical value is set as 160, which is 16 times larger than the initial one to simulate the steep slope of workload change. Time slot 1 is from time 30 to time 130, time slot 2 is from time 150 to time 210, and time slot 3 is from time 235 to time 280. Time slot 1 is the increment of constant growth, time slot 2 and time slot 3 are the increment of exponential growth.

In order to better demonstrating experimental results, we develop two elastic databases' frameworks: first is the OPEN elastic database which does not use Deep Q Learning. The OPEN elastic database only runs scale out and scale in rules when the response time exceeds the Service Level Agreement (SLA). This is a passive adaption to the change of the environment. In our experiment, the OPEN elastic database mainly plays a baseline role for comparison; second is our target framework self-configured elastic database with the implement of Deep Q Learning. It takes the initiative in configuring VMs' parameters to adapt the change of the environment.

4.2 External System Performance

After sufficient experience learning, experimental results demonstrate that a self-configured elastic database can always make an optimal action. From the comparison shown in Fig-

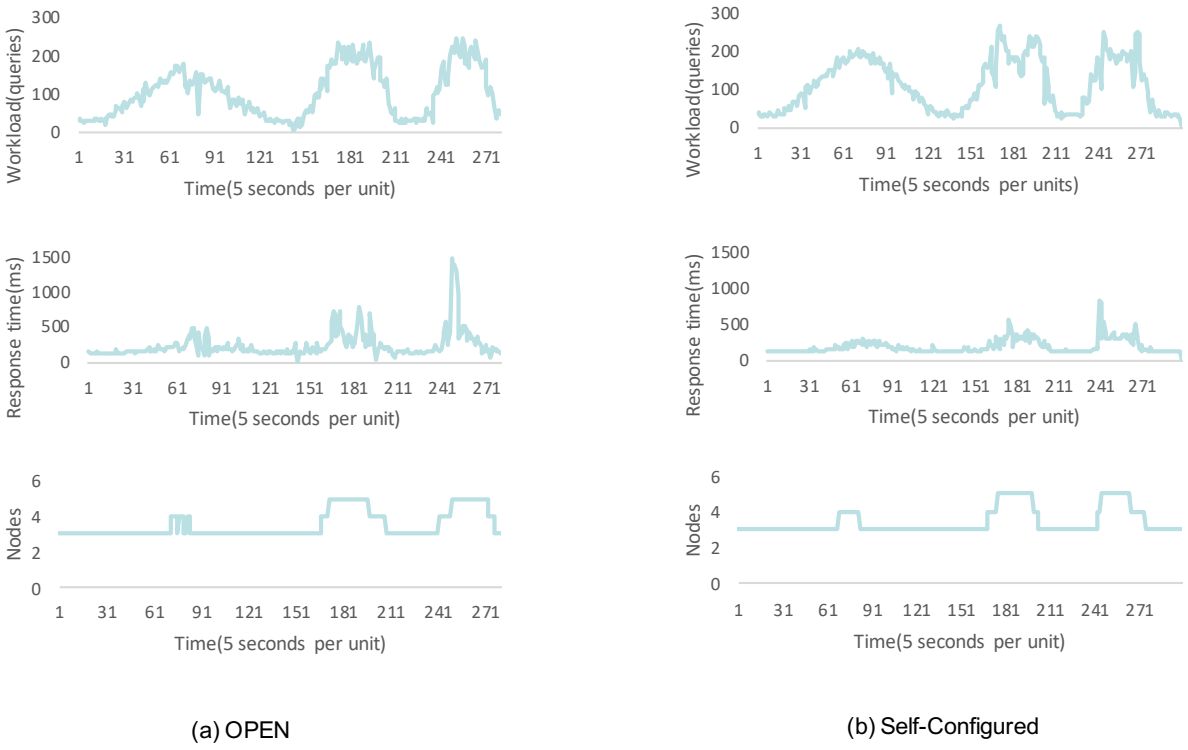

Figure 4.2 Performance comparison between Self-Configured and OPEN.

ure 4.2, which monitors workload, response time and the number of raised servers, Figure 4.2(b) shows that the response time in self-configured elastic database is continuously kept in the range of SLA. Compared to Figure 4.2(a) corresponding to the OPEN elastic database, the self-configured elastic database is able to reduce up to 90% SLA violations. During the period of time 70 to time 90, the OPEN elastic database executes scale out and scale in rules frequently for the unstable response time. In such passivity OPEN elastic database, it a terrible stress reaction which will increase system load and waste system resources. Generally speaking, in a real world environment system, workload fluctuations are inevitable, but in self-configured elastic database, with the penalty of the reward function, Deep Q Learning is able to bypass such inefficient action. From the Figure 4.2(b), we can see that a self-configured elastic database could consistently choose an optimal policy to drive the environment into a high-effective configuration state. Even though a self-configured elastic database reveals a SLA violation in some certain arduous conditions, the variances in response time are significantly decreased, which means the system using DQL is more stable. This suggests that self-configured elastic database enables system efficient robustness toward complicated environment.

4.3 Internal System Performance

Figure 4.2 demonstrates a high-productive external performance in a self-configured elastic database, profitable internal resource utilization is another advantage in the self-configured elastic database. In our experiment, we use a resourceful tool *Dstat* to monitor servers performance in real-time. CPU is the major bottleneck in our elastic environment, so we mainly monitor the usage of CPU. Figure 4.3 is the CPU usage of the OPEN elastic database, and Figure 4.4 is the CPU usage of the self-configured elastic database. For the reason that in a self-configured elastic database, the minimum control unit is down to each core of the CPU. Therefore, CPU utilization improves significantly. At the beginning of both

elastic databases, the initial state is one master server and two slaves servers with all default CPU settings. During the time period before changing workload, the self-configured elastic database diagnoses the system that it is not necessary to keep full CPU power, so it gradually reduced the core of each CPU and then the CPU utilization increases adaptively. During the period of workload change, the CPU utilization in the self-configured elastic database fluctuates up and down quickly. This is caused by in order to achieving high resource utilization, control units in the self-configured elastic database were much more sensitive to the workload change than in the OPEN elastic database. However, according to the reward function and the self-configured exploration policy, the CPU usage can still maintain in a rational range. Compared to the OPEN elastic database, whose CPU usage is highly related to workload change, the CPU usage of the self-configured elastic database has a weak affiliation with the workload input. Thus, it enables more efficient generalization ability to the system.

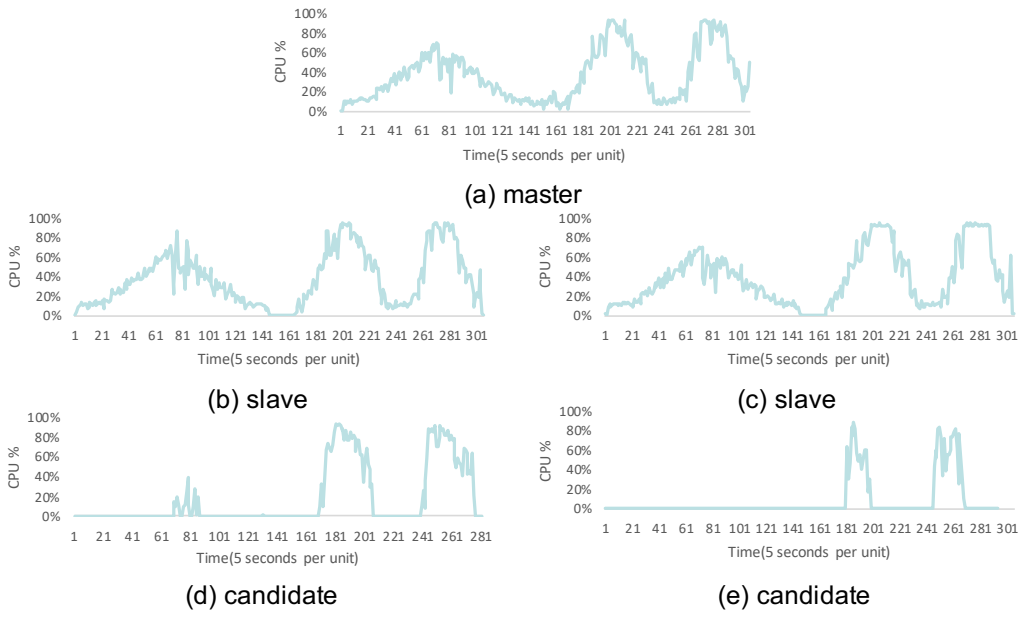

Figure 4.3 OPEN CPU usage.

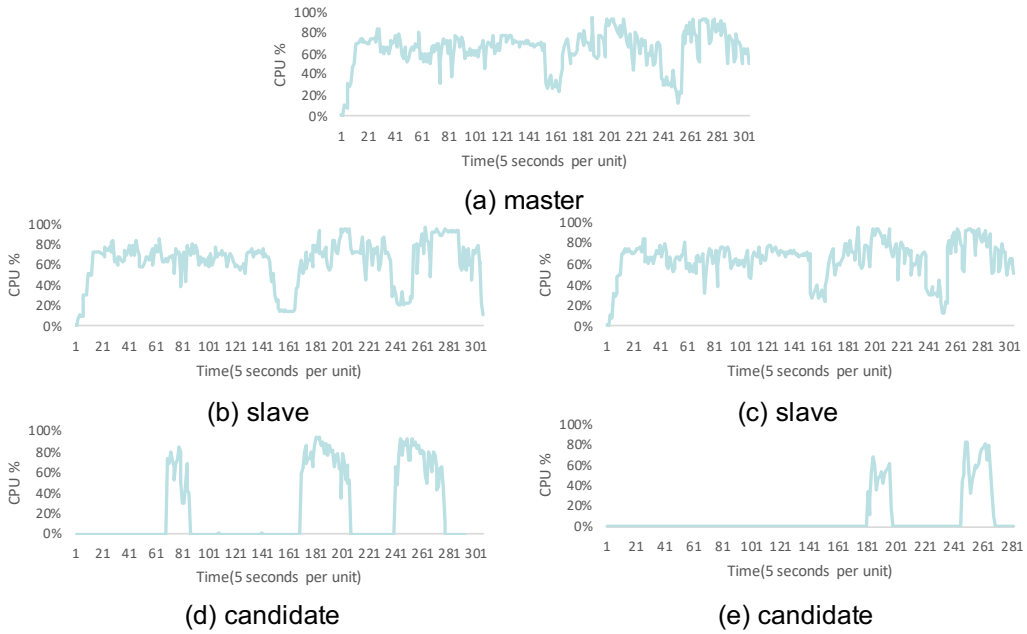

Figure 4.4 Self-configured CPU usage.

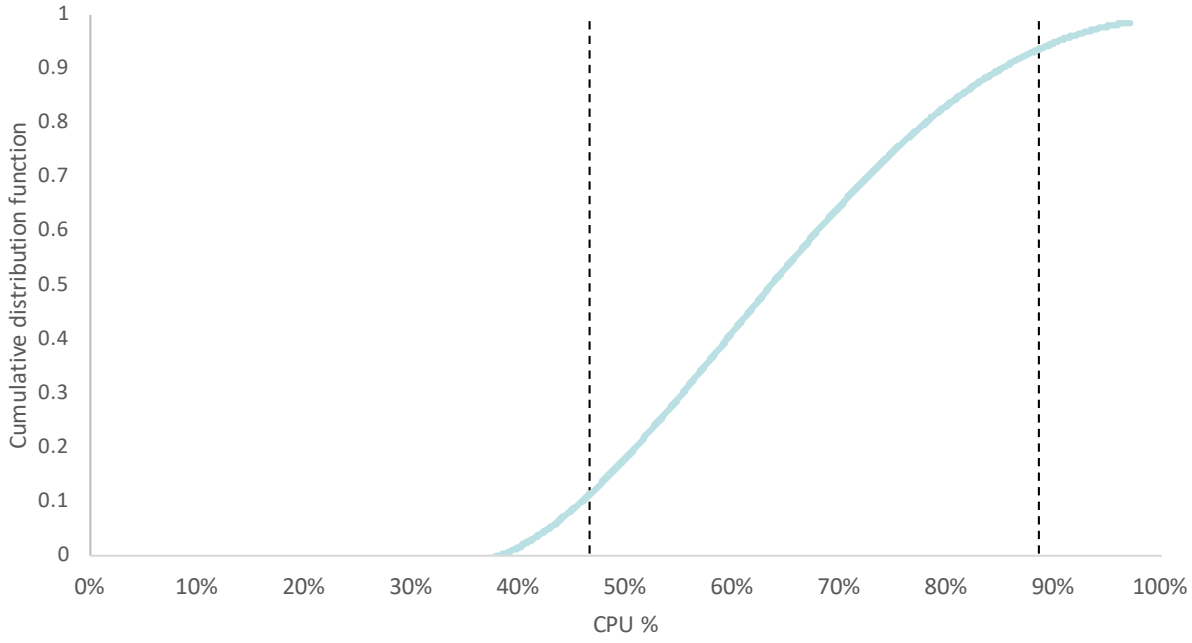

Figure 4.5 CPU utilization distribution.

4.4 Effect of Self-configured Exploration

Figure 4.5 is the cumulative distribution function (CDF) of CPU utilization in one master server and two slave servers. Candidate servers are excluded because they are unrepresentative for the lack of data. In our experiment, we set $U_{cpu}^{ub} = 90\%$ and $U_{cpu}^{lb} = 50\%$, respectively. During the process of collecting the data, we exclude warm up, warm down periods and the intervals between experiment time slots. Therefore, we can see that, the outlier point of CPU utilization is below 40%. Through the boundary of self-configured exploration, the CPU utilization is mainly concentrated in 50% to 90% letting the self-configured elastic database to a higher learning efficiency.

CHAPTER 5. CONCLUSIONS AND FUTURE CHALLENGE

In this self-configured elastic database, DQL shows its excellent ability in resource management. With the combination of the Deep Learning and Reinforcement Learning, a self-configured elastic database has a great potential in dealing with high dimensional problem and excellent evaluation mechanism to learn the interface between databases and environment. Furthermore, with the sufficient training experience, self-configured elastic database can efficiently reduce SLA violation that helps the system smoothly pull through the challenge of a big boost in users' workload.

However, there are some challenges that need to be addressed in the future. First, the sequence of workload input to the client emulator layer is self-defined, so it has a limitation that parameters of DQL model only achieve good performance in the environment we provided. Second, although DQL model is used for processing high-dimensional data, our experiments only consider CPU configuration parameters, while on the AWS platform, we assumed CPU is the major bottleneck to elastic databases' performance, there exist a number of bottlenecks that violate that assumption. Therefore, further research is needed to develop the DQL model with more training data and implement more configuration parameters to inspire DQL potential value.

REFERENCES

[1] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM.

[2] Bu, X., Rao, J., and Xu, C.-Z. (2013). Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Transactions on Parallel and Distributed Systems*, 24(4):681–690.

[3] Campbell, D. G., Kakivaya, G., and Ellis, N. (2010). Extreme scale with full SQL language support in microsoft SQL azure. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1021–1024. ACM.

[4] Cardellini, V., Colajanni, M., and Yu, P. S. (1999). Dynamic load balancing on Web-server systems. *IEEE Internet computing*, 3(3):28–39.

[5] Caton, S., Haas, C., Chard, K., Bubendorfer, K., and Rana, O. F. (2014). A social compute cloud: Allocating and sharing infrastructure resources via social networks. *IEEE Transactions on Services Computing*, 7(3):359–372.

[6] Cattell, R. (2011). Scalable SQL and NoSQL data stores. *ACM Sigmod Record*, 39(4):12–27.

[7] Das, S., El Abbadi, A., and Agrawal, D. (2009). Elastras: An elastic transactional data store in the cloud. *HotCloud*, 9:131–142.

[8] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. 41(6):205–220.

[9] García, D. F. and García, J. (2003). TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48.

[10] Helland, P. (2007). Life beyond distributed transactions: an apostate's opinion. In *CIDR*, volume 2007, pages 132–141.

[11] Lakshman, A. and Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40.

[12] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937.

[13] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[14] Moon, H. J., Hacıgümüş, H., Chi, Y., and Hsiung, W.-P. (2013). Swat: a lightweight load balancing method for multitenant databases. In *Proceedings of the 16th International Conference on Extending Database Technology*, pages 65–76. ACM.

[15] Pavlo, A., Curino, C., and Zdonik, S. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 61–72. ACM.

[16] Roy, N., Dubey, A., and Gokhale, A. (2011). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE.

[17] Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

[18] Serafini, M., Mansour, E., Aboulnaga, A., Salem, K., Rafiq, T., and Minhas, U. F. (2014). Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12):1035–1046.

[19] Shreedhar, M. and Varghese, G. (1996). Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, (3):375–385.

[20] Smith, C. 150 amazing Amazon statistics and facts. https://expandedramblings.com/index.php/amazon-statistics.

[21] Sutton, R. S., Barto, A. G., et al. (1998). *Introduction to reinforcement learning*, volume 135. MIT press Cambridge.

[22] Tokic, M. (2010). Adaptive ε-greedy exploration in reinforcement learning based on value differences. In *Annual Conference on Artificial Intelligence*, pages 203–210. Springer.

[23] Wang, W., Li, B., and Liang, B. (2014). Dominant resource fairness in cloud computing systems with heterogeneous servers. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 583–591. IEEE.

[24] Wang, Y., Wang, X., Chen, M., and Zhu, X. (2008). Power-efficient response time guarantees for virtualized enterprise servers. In *2008 Real-Time Systems Symposium*, pages 303–312. IEEE.

[25] Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.

[26] Wikipedia. ACID (computer science). https://en.wikipedia.org/wiki/ACID_(computer_science).

[27] Xiao, Z., Song, W., and Chen, Q. (2013). Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117.

[28] Zawodny, J. D. and Balling, D. J. (2004). *High Performance MySQL: Optimization, Backups, Replication, Load Balancing & More.* " O'Reilly Media, Inc.".

[29] Zhang, Q., Zhani, M. F., Boutaba, R., and Hellerstein, J. L. (2014). Dynamic heterogeneity-aware resource provisioning in the cloud. *IEEE transactions on cloud computing*, 2(1):14–28.

[30] Zhang, S., Zhang, S., Chen, X., and Huo, X. (2010). Cloud computing research and development trend. In *2010 Second international conference on future networks*, pages 93–97. Ieee.